Improved and Extended Locating Functionality on Compressed Suffix Arrays[☆]

Simon Gog[a], Gonzalo Navarro[b], Matthias Petri[c]

[a] *Institute of Theoretical Informatics, Karlsruhe Institute of Technology, Germany*
[b] *Center of Biotechnology and Bioengineering, Department of Computer Science, University of Chile, Chile*
[c] *Department of Computing and Informations Systems, The University of Melbourne, Australia*

Abstract

Compressed Suffix Arrays (CSAs) offer the same functionality as classical suffix arrays (SAs), and more, within space close to that of the compressed text, and in addition they can reproduce any text fragment. Furthermore, their pattern search times are comparable to those of SAs. This combination has made CSAs extremely successful substitutes for SAs on space-demanding applications. Their weakest point is that they are orders of magnitude slower when retrieving the precise positions of pattern occurrences. SAs have other well-known shortcomings, inherited by CSAs, such as not retrieving those positions in a useful order. In this paper we present new techniques that, on the one hand, improve the current space/time tradeoffs for retrieving pattern occurrences in CSAs, and on the other, efficiently support extended pattern locating functionalities, such as retrieving occurrences in text order or limiting the occurrences to within a text window. Our experimental results display considerable savings with respect to the baseline techniques in many cases of interest: in some cases we outperform their time by a factor of two or three.

1. Introduction

Suffix arrays [16, 28] are text indexing data structures that support various pattern matching functionalities. Built on a text $T[1,n]$ over an alphabet $[1,\sigma]$, the most basic functionality provided by a suffix array (SA) is to *count* the number of times a given pattern $P[1,m]$ appears in T. This can be done in $O(m \log n)$ and even $O(m + \log n)$ time [28]. Once counted, SAs can retrieve each of the occ positions of P in T in $O(1)$ time (this is called *reporting* or

[☆] An early partial version of this article appeared in *Proc. SEA 2014* [14]. Funded with basal funds FB0001, Conicyt, Chile and ARC grants DP110101743 and DP140103256, Australia.

Email addresses: gog@kit.edu (Simon Gog), gnavarro@dcc.uchile.cl (Gonzalo Navarro), matthias.petri@unimelb.edu.au (Matthias Petri)

locating the pattern occurrences). A suffix array uses $O(n \log n)$ *bits* of space and can be built in $O(n)$ time [25, 24, 23].

The space usage of suffix arrays, albeit "linear" in classical terms, is asymptotically larger than the $n \lg \sigma$ bits needed to represent T itself.[1] Since the year 2000, two families of compressed suffix arrays (CSAs) have emerged [32]. One family, simply called CSAs [19, 20, 38, 39, 18], is built on the compressibility of a so-called Ψ function (see details in the next section), and simulates the basic SA procedure for pattern searching, achieving the same $O(m \log n)$ counting time of basic SAs. A second family, called FM-indexes [6, 7, 8, 1], built on the Burrows-Wheeler transform [3] of T and on a new concept called backward-search, which allows $O(m \log \sigma)$ and even $O(m)$ time for counting occurrences. The counting times of all CSAs are comparable to those of SAs in practical terms as well [5]. Their space usage can be made asymptotically equal to that of the *compressed* text under the k-th order empirical entropy model, and in all cases it is $O(n \log \sigma)$ bits. Within this space, CSAs support even stronger functionalities than SAs. In particular, they can reconstruct any text segment $T[l, r]$, as well as to compute "inverse" suffix array entries (again, details in the next section), efficiently. Reproducing any text segment allows CSAs to *replace* T, further reducing space.

The weakest part of CSAs in general is that they are much slower than SAs at retrieving the *occ* positions where P occurs in T. SAs require basically *occ* contiguous memory accesses. Instead, both CSA families use a sampling parameter s that induces an extra space of $O((n/s) \log n)$ bits (and therefore s is typically chosen to be $\Omega(\log n)$); then Ψ-based CSAs require $O(s)$ time per reported position and FM-indexes require $O(s \log \sigma)$. In practical terms, all CSAs are orders of magnitude slower than SAs when reporting occurrence positions [5], even when the distribution of the queries is known [10]. Text extraction complexities for windows $T[l, r]$ are also affected by s, but to a lesser degree: they require $O(s + r - l)$ steps.

Although widely acknowledged as a powerful and flexible tool for text searching activities, the SA has some drawbacks that can be problematic in certain applications. The simplest one is that it retrieves the occurrence positions of P in an order that is not left-to-right in the text. This complicates displaying the occurrences in order (unless one obtains and sorts them all), as for example when displaying the occurrences progressively in a document viewer. A related one is that there is no efficient way to retrieve only the occurrences of P that are within a window of T unless one uses $\Omega(n \log n)$ bits of space [27, 2, 33, 21]. This is useful, for example, to display occurrences only within some documents of a collection (T being the concatenation of the documents), for instance only recent news in a collection of news documents.

In this paper we present new techniques that speed up the basic pattern locating functionalities of CSAs, and also efficiently support extended functionalities. Our experimental results show that the new solutions outperform the

[1] We use lg to denote logarithm to the base 2.

baseline solutions, in many cases of interest, by a wide margin. The detailed breakdown of our contributions is as follows.

1. We unify the samplings for pattern locating and for displaying text substrings into a single data structure, by using the fact that they are essentially inverse permutations of each other. This yields improved space/time tradeoffs for locating pattern positions and displaying text substrings, especially in memory-reduced scenarios where large values of s must be used.
2. We show that CSAs, which are based on the Ψ function, can be further improved by using two methods of representing increasing lists, which were recently applied successfully to inverted indexes [44, 35]. Our experiments show that adapting these techniques to CSAs results in improved space/time tradeoffs for small and large alphabet inputs. For example, for a word parsed text, the new solutions are more than twice as fast as previous state-of-the-art CSA implementations.
3. The *occ* positions of P have variable locating cost in a CSA. We use a data structure that takes $2n+o(n)$ additional bits to report the occurrences of P from cheapest to most expensive, thereby making reporting considerably faster when only some occurrences must be displayed (as in search engine interfaces, or when one displays results progressively and can show a few and process the rest in the background). Our experiments show that, when reporting less than around 15% of the occurrences, this technique is faster than reporting random occurrences, even when the baseline uses those extra $2n + o(n)$ bits to reduce s. A simple alternative that turns out to be very competitive is just to report first the occurrences that are sampled in the CSA, and thus can be reported at basically no cost.
4. Variants of the previous idea have been used for document listing [31] and for reporting positions in text order [33]. We study this latter application in practice. While for showing *all* the occurrences in order it is better to extract and partially sort them, one might need to show only the first occurrences, or might have to show the occurrences progressively. Our implementation becomes faster than the baseline when we report a fraction below 25% of the occurrences, and improves for lower fractions. For example, we report three times faster the first 5% of the occurrences, even letting the baseline spend those $2n+o(n)$ extra bits on a denser sampling.
5. Finally, we extend this second idea to report the text positions that are within a given text window. While the result is not competitive for windows located at random positions of T, our method is faster than the baseline of filtering the text positions by brute force when the window is within the first 15% of T. This is particularly useful in versioned collections or news archives, when the latest versions/dates are those most frequently queried.

The improved sampling we propose is now available in the Succinct Data Structure Library (SDSL). The library contains state-of-the-art `C++11` implementations of many succinct data structures proposed in over 40 research publications. It is available in at https://github.com/simongog/sdsl-lite.

2. Compressed Suffix Arrays

Let $T[0, n-1]$ be a text over the alphabet $[0, \sigma - 1]$. Then a substring $T[i, n-1]$ is called a *suffix* of T, and is identified with position i. A *suffix array* $SA[0, n-1]$ is a permutation of $[0, n-1]$ containing the positions of the n suffixes of T in increasing lexicographic order (thus the suffix array uses at least $n \lg n$ bits). Since the positions of the occurrences of $P[0, m-1]$ in T are precisely the suffixes of T that start with P, and those form a lexicographic range, *counting* the number of occurrences of P in T is done via two binary searches using SA and T, within $O(m \log n)$ time. Once we find that $SA[sp, ep]$ contains all the occurrences of P in T, their number is $occ = ep - sp + 1$ and their positions are $SA[sp], SA[sp+1], \ldots, SA[ep]$. With some further structures adding up to $O(n \log n)$ bits, suffix arrays can do the counting in $O(m + \log n)$ time [28]. This can be reduced to $O(m)$ by resorting to suffix trees [43], which still use $O(n \log n)$ bits but too much space in practice.

Our interest in this paper is precisely using less space while retaining the SA functionality. A *compressed suffix array* (CSA) is a data structure that emulates the SA while using $O(n \log \sigma)$ bits of space, and usually less on compressible texts [32]. One family of CSAs [19, 20, 38, 39, 18] builds on the so-called Ψ function: $\Psi(i) = SA^{-1}[SA[i] + 1]$, where SA^{-1} is the inverse permutation of the suffix array (given a text position j, $SA^{-1}[j]$ tells where in the suffix array is the pointer to the suffix $T[j, n-1]$). Thus, if $SA[i] = j$, $\Psi(i)$ tells where is $j+1$ mentioned in SA, $SA[\Psi(i)] = SA[i] + 1 = j + 1$. It turns out that array Ψ is compressible up to the k-th order empirical entropy of T [29]. With small additional data structures, Ψ-based CSAs find the range $[sp, ep]$ for $P[0, m-1]$ in $O(m \log n)$ time.

A second family, FM-indexes [6, 7, 8, 1], build on the Burrows-Wheeler transform [3] (BWT) of T, denoted T^{bwt}, which is a reversible permutation of the symbols in T that turns out to be easier to compress. With light extra structures on top of T^{bwt}, one can implement a function called $LF(i) = SA^{-1}[SA[i] - 1]$, the inverse of Ψ, in time at most $O(\log \sigma)$. An extension to the LF function is used to implement a so-called backward-search, which allows finding the interval $[sp, ep]$ corresponding to a pattern $P[0, m-1]$ in $O(m \log \sigma)$ and even $O(m)$ time [1]. Some Ψ-based CSAs have also been adapted to backward search, retaining their $O(m \log n)$ search complexity but reducing time in practice [39].

Once the range $SA[sp, ep]$ is found (and hence the counting problem is solved), locating the occurrences of P requires finding out the values of $SA[k]$ for $k \in [sp, ep]$, which are not directly stored in CSAs. All the practical CSAs use essentially the same solution for locating occurrences [32]. Text T is sampled at regular intervals of length s, and we store those sampled text positions in a *sampled suffix array* $SA_s[0, n/s]$, in suffix array order. More precisely, we mark in a bitmap $B[0, n-1]$ the positions $SA^{-1}[s \cdot j]$, for all j, with a 1, and the rest are 0s. Now we traverse B left to right, and append the value $SA[i]/s$ to SA_s for each i such that $B[i] = 1$. Array SA_s requires $(n/s) \lg(n/s) + O(n/s)$ bits of space, and B can be implemented in compressed form using $(n/s) \lg s + O(n/s) + o(n)$ bits [36, 34], for a total of $(n/s) \lg n + O(n/s) + o(n)$ bits.

To locate $SA[i]$, we proceed as follows on a Ψ-based CSA. If $B[i] = 1$, then the position is sampled and we know its value is in SA_s, precisely at position $rank_1(B, i)$, which counts the number of 1s in $B[1, i]$ (this function is implemented in constant time in the compressed representation of B [36]). Otherwise, we test $B[\Psi(i)]$, $B[\Psi^2(i)]$, and so on until we find $B[\Psi^k(i)] = B[i'] = 1$. Then we find the corresponding value at SA_s; the final answer is $SA[i] = SA_s[rank_1(B, i')] \cdot s - k$. The procedure is analogous on an FM-index, using function LF, which traverses T backwards instead of forwards. The sampling guarantees that we need to perform at most s steps before obtaining $SA[i]$.

To display $T[l, r]$ we use the same sampling positions $s \cdot j$, and store a *sampled inverse suffix array* $SA_s^{-1}[1, n/s]$ with the suffix array positions that point to the sampled text positions, in text order. More precisely, we store $SA_s^{-1}[j] = SA^{-1}[j \cdot s]$ for all j. This requires other $(n/s) \lg s + O(n/s)$ bits of space. Then, in order to display $T[l, r]$ with a Ψ-based CSA, we start displaying slightly earlier, at text position $\lfloor l/s \rfloor \cdot s$, which is pointed from position $i = SA_s^{-1}[\lfloor l/s \rfloor]$ in SA. The first letter of a suffix $SA[i]$ is easily obtained on all CSAs if i is known. Therefore, displaying is done by listing the first letter of suffixes pointed from $SA[i], SA[\Psi(i)], SA[\Psi^2(i)], \ldots$ until covering the window $T[l, r]$. The process is analogous on FM-indexes. In total, we need at most $s + r - l$ steps.

This mechanism is useful as well to compute any $SA^{-1}[j]$ value. If j is a multiple of s then the answer is at $SA_s^{-1}[j/s]$. Otherwise, on a Ψ-based CSA, we start at $i = SA_s^{-1}[\lfloor j/s \rfloor]$ and the answer is $\Psi^k(i)$, for $k = j - \lfloor j/s \rfloor \cdot s$ (analogously on an FM-index), taking up to s steps. Computing $SA^{-1}[j]$ is useful in many scenarios, such as compressed suffix trees [40, 12] and document retrieval [41].

3. A Combined Structure for Locating and Displaying

In order to have a performance related to s in locating and displaying text, the basic scheme uses $2(n/s) \lg n + O(n/s) + o(n)$ bits. In this section we show that this can be reduced to $(1 + \epsilon)(n/s) \lg n + O(n/s) + o(n)$ bits, retaining the same locating cost and increasing the display cost to just $1/\epsilon + s + r - l$ steps.

The key is to realize that the SA_s and SA_s^{-1} are essentially inverse permutations of each other. Assume we store, instead of the value $i = SA_s^{-1}[j]$, the smaller value $i' = SA_s^{-1*}[j] = rank_1(B, i)$. Since $B[i] = 1$, we can retrieve i from i' with the operation $i = select_1(B, i')$, which finds the i'th 1 in B and is implemented in constant time in the compressed representation of B [36]. Now, at the cost of computing $select_1$ once when displaying a text range, we can store SA_s^{-1*} in $(n/s) \lg(n/s) + O(n/s)$ bits. What is more important, however, is that SA_s and SA_s^{-1*} arrays are two permutations on $[0, n/s]$, and are inverses of each other. Fig. 1 shows an example.

Theorem 1. *Permutations SA_s and SA_s^{-1*} are inverses of each other.*

Proof. $SA_s[SA_s^{-1*}[j]] = SA_s[rank_1(B, SA_s^{-1}[j])] = SA[SA_s^{-1}[j]]/s = SA[SA^{-1}[j \cdot s]]/s = (j \cdot s)/s = j$.

i	SA	SA^{-1}	B	T
0	13	**6**	0	$
1	**12**	7	**1**	a$
2	4	9	0	atenatsea$
3	8	**5**	0	atsea$
4	11	2	0	ea$
5	**3**	12	**1**	eatenatsea$
6	**0**	**8**	**1**	eeleatenatsea$
7	1	10	0	eleatenatsea$
8	**6**	3	**1**	enatsea$
9	2	**13**	0	leatenatsea$
10	7	11	0	natsea$
11	10	4	0	sea$
12	5	**1**	0	tenatsea$
13	**9**	0	**1**	tsea$

$SA_s = \quad 4 \quad 1 \quad 0 \quad 2 \quad 3$

$SA_s^{-1} = \quad 6 \quad 5 \quad 8 \quad 13 \quad 1$

$SA_s^{-1*} = \quad 2 \quad 1 \quad 3 \quad 4 \quad 0$

$$SA^{-1}[i \cdot s] = SA_s^{-1}[i] = select(B, SA_s^{-1*}[i])$$

Figure 1: Example of a suffix array, its inverse, and the sample arrays SA_s, the conceptional and formerly used SA^{-1}, and SA^{-1*} for $s = 3$.

Munro et al. [30] showed how to store a permutation $\pi[0, n' - 1]$ in $(1 + \epsilon)n' \lg n' + O(n')$ bits, so that any $\pi(i)$ can be computed in constant time and any $\pi^{-1}(j)$ in time $O(1/\epsilon)$. Basically, they add a data structure using $\epsilon\, n' \lg n' + O(n')$ bits on top of a plain array storing π. By applying this technique to SA_s, we retain the same fast access time to it, while obtaining $O(1/\epsilon)$ time access to SA_s^{-1*} without the need to represent it directly. This yields the promised result (more precisely, the space is $(1 + \epsilon)(n/s) \lg(n/s) + (n/s) \lg s + O(n/s) + o(n)$ bits). We choose to retain faster access to SA_s because it is more frequently used, and for displaying the extra $O(1/\epsilon)$ time cost is blurred by the remaining $O(s + r - l)$ time. One is free, of course, to choose the opposite.

Our experiments will show that this technique is practical and yields a significant improvement in the space-time tradeoff of CSAs, especially when the space is scarce and relatively large s values must be used.

4. Faster Locating on Ψ-based CSAs

CSAs based on Ψ built over a sequence of integer tokens are remarkably similar to inverted indexes. Inverted indexes are generally built over a tokenized and stemmed representation of a given input text. In an inverted index, each unique parsed token t is represented (among other components) by a postings list consisting of an increasing sequence of document identifiers containing t. Many storage schemes have been proposed to efficiently compress and process these increasing sequences of integers.

Array Ψ is the concatenation of σ increasing sequences, one for each of the σ symbols in the input. Those sequences do not represent document identifiers, but rather suffix array positions. Usually, the values in Ψ are differentially encoded using Elias-γ/δ codes [39]. The codes have been engineered to allow efficient decoding using lookup tables [22].

In practice, the complete Ψ array is encoded as a continuous sequence of integers. The sequence is split into blocks of equal size (e.g., 128 elements) –

to allow random access into Ψ – which are then differentially encoded using Elias codes. A transition between two runs of adjacent symbols in Ψ within a block can result in difference encoding generating a negative number. Instead of using a "run aware" encoding scheme where differences between symbols in adjacent runs are not computed, the negative numbers are encoded as large positive numbers. This simplifies the encoding and decoding process and avoids storing additional information to identify run transitions in the encoded data.

Here we explore how recent advances in compression techniques used for postings list compression can be applied to Ψ-based CSAs. Specifically, we explore the applicability of PForDelta based compression codes [44] and partitioned Elias-Fano inspired encodings [26] to compress Ψ.

Replacing Elias-γ/δ coding by PForDelta. The PForDelta family of compression codes encodes differences between adjacent integers using a fixed number of b bits; numbers needing more bits are encoded separately as exceptions. Different schemes exist which employ different strategies to choose b and to encode the exceptions. In practice, the PForDelta scheme providing the best compression effectiveness [26] is the `OptPFor` scheme of Yan et al. [44]. `OptPFor` chooses b optimally for each encoded block and uses `Simple16` [45] to encode exceptions. Lemire and Boytsov [26] further employ SIMD instructions to extract b integers efficiently before decoding the exceptions of each block. However, many of the SIMD schemes are currently restricted to 32-bit integers. Adapting `OptPFor` to encode and decode Ψ is straightforward. Similarly to Elias based codes, the sequence is split into blocks of fixed size. The differences within a block are encoded using `OptPFor`. Potential negative values resulting from difference encoding across run boundaries are again represented as large positive values. Within a block, this value will then be encoded as an exception, whereas smaller values within each block can still be encoded efficiently.

Adapting partitioned Elias-Fano-Coding. Recently, Ottaviano and Venturini [35] have proposed a block based alternative encoding scheme based on the Elias-Fano [4] representation of monotone sequences, instead of standard differential based encoding schemes. They extend the Elias-Fano based postings list encoding scheme of Vigna [42] by creating a two-level structure for each postings list. The top level structure stores an Elias-Fano encoded sequence of block representatives, which can be used to efficiently find numbers in postings lists (a basic operation in many inverted index query processing schemes). The bottom level of the postings list is encoded by partitioning the integer sequence into blocks, where different encoding schemes are used for each block. Partitioning is either performed uniformly into fixed sized blocks, or "optimally" using an approximation scheme [9]. Individual blocks are stored using three different encoding schemes: (1) as a plain bitvector (2) as an Elias-Fano encoded sequence or (3) no encoding. The latter is used if the block contains all the values in between two block representatives stored in the top level. For example, a block consisting of all 128 values in $[257, 385]$ can be encoded using zero bits, as the

contents can be inferred from the values stored in the top level: 256, the last value in the previous block, and 385. Such blocks are called uniform.

Adapting this partitioned Elias-Fano based encoding scheme to encode Ψ requires certain modifications to the original scheme [35]. Instead of encoding individual monotonically increasing sequences, a sequence containing σ monotonically increasing runs has to be encoded. Thus, certain properties used by Ottaviano and Venturini do not hold in our case: (1) the top level structure storing the block representatives is not monotonically increasing; (2) the top level structure cannot be used to determine if a block is uniform; (3) individual blocks in the lower level may not contain a monotonically increasing sequence. Vigna [42] encodes frequency information in inverted indexes using Elias-Fano encoding by storing the prefix-sum of the values in a non-increasing sequence. In case individual blocks contain run transitions, of which there are σ in total, we store the block as a prefix-sum encoded sequence. For each block we store, in a bitvector, if the block is prefix-sum encoded or if it contains a monotonically increasing sequence where the regular partition Elias-Fano scheme [35] is used. Note that this cannot be determined using the top level entries, as a run transition may occur within a block, whereas the top level entries for the adjacent blocks remain increasing. Additionally, Ottaviano and Venturini subtract the largest value in the previous block (stored in the top level) from all values in the current block to decrease the universe within which block values have to be encoded. This, again, cannot always be applied: While the values in a block might be monotonically increasing, the last value in the previous block might be the last value of the previous run, and can thus be larger than the smallest value in the current block. Encoding the top level sequence also provides several trade-offs. To allow fast search within individual runs, it may be useful to encode the non-increasing sequences using prefix-sum based Elias-Fano encoding. Ottaviano and Venturini use fast CPU instructions to sequentially process the Elias-Fano sequence; to allow faster random access to sequence it might be beneficial to support constant time `select` operations on the high parts of the top level bitvector [34].

Uniform Blocks. The notion of uniform blocks used in the partitioned Elias-Fano encoding scheme can also be applied to the `OptPFor` and Elias-γ/δ schemes. If the block contains all values within the block interval, no encoding is necessary. While this optimization seems to be straightforward it was not used in the original CSA implementation of Sadakane [38] or in previous versions of SDSL [13].

Experiments. In the experimental section we will explore the performance of different Ψ encoding methods. We compare standard Elias-δ based encoding[2] to `OptPFor` and Elias-Fano based encoding schemes. We measure the performance of accessing $\Psi[i]$ and how differences in $\Psi[i]$ access performance translate to

[2]The original CSA implementation by Sadakane uses Elias-γ, but both performed similarly in our experiments.

accessing $SA[i]$ and $SA^{-1}[i]$ respectively. We further evaluate the performance over both small and large alphabet inputs as alphabet size influences the number of run transitions within Ψ.

5. A Structure for Prioritized Location of Occurrences

Assume we want to locate only some, say t, occurrences of P in $SA[sp, ep]$, as for example in many interfaces that show a few results. In a Ψ-based CSA, the number of steps needed to compute $SA[k]$ is its distance to the next multiple of s, that is, $D[k] = 0$ if $SA[k]$ is a multiple of s and $D[k] = s - (SA[k] \bmod s)$ otherwise. In an FM-index, the cost is $D[k] = SA[k] \bmod s$. We would like to use this information to choose low-cost entries $k \in [sp, ep]$ instead of arbitrary ones, as current CSAs do. The problem, of course, is that we do not yet know the value of $SA[k]$ before computing it! (some CSAs actually store the value $SA[k] \bmod s$ [37], but they are not competitive with the best current CSAs when using the same space [17]).

However, we do not need that much information. It is sufficient to know, given a range $D[x, y]$, *which is the minimum value* in that range. This is called a *range minimum query* (RMQ): $\text{RMQ}(x, y)$ is the position of a minimum value in $D[x, y]$. The best current (and optimal) result for RMQs [11] preprocesses D in linear time, producing a data structure that uses just $2n + o(n)$ bits. Then the structure, *without accessing* D, can answer $\text{RMQ}(x, y)$ queries in $O(1)$ time.

The basic method. We use the RMQ data structure to progressively obtain the values of $SA[sp, ep]$ from cheapest to most expensive, as follows [31]. We compute $k = \text{RMQ}(sp, ep)$, which is the cheapest value in the range, retrieve and report $SA[k]$ as our first value, and compute $D[k]$ from it. The tuple $\langle sp, ep, k, D[k]\rangle$ is inserted as the first element in a min-priority queue that sorts the tuples by $D[k]$ values. Now we iteratively extract the first (cheapest) element from the queue, let it be the tuple $\langle lp, rp, k, v\rangle$, compute $k_l = \text{RMQ}(lp, k-1)$ and $k_r = \text{RMQ}(k+1, rp)$, then retrieve and report $SA[k_l]$ and $SA[k_r]$, and insert tuples $\langle lp, k-1, k_l, D[k]\rangle$ and $\langle k+1, rp, k_r, D[k]\rangle$ in the priority queue (unless they are empty intervals). We stop when we have extracted the desired number of answers or when the queue becomes empty. We carry out $O(t)$ steps to report t occurrences [31].

A stronger solution using B. Recall bitmap B that marks the sampled positions. The places where $D[k] = 0$ are precisely those where $B[k] = 1$. We can use this to slightly reduce the space of the data structure. First, using operations $rank_1$ and $select_1$ on B, we spot all those $k \in [sp, ep]$ where $B[k] = 1$. Only then we start reporting the next cheapest occurrences using the RMQ data structure as above. This structure, however, is built only on the entries of array D', which contains all $D[k] \neq 0$. Using $rank_0$ operations on B (which counts 0s, $rank_0(B, i) = i - rank_1(B, i)$), we map positions in $D[lp, rp]$ to $D'[lp', rp']$. Mapping back can be solved by using a $select_0$ structure on B, but we opt for an alternative that is faster in practice and spends little extra memory: we create

a sorted list of pairs $\langle k, rank_0(B,k)\rangle$ for the already spotted k with $B[k]=1$, and binary search it for mapping the positions back.

Refining priorities. The process can be further optimized by refining the ordering of the priority queue. Our method sorts the intervals $[sp,ep]$ only according to the minimum possible value u ($= D[k]$). Assuming that the values in $D[sp,ep]$ are distributed uniformly at random in $[u,s)$ we can calculate the value of the expected minimum $\eta(lp,rp,u) = u + \sum_{v=u+1}^{s-1} \left(\frac{s-v}{s-u}\right)^z$, where $z = rp-lp+1$ is the range size. This can be used as a refined priority value.

Experiments. In the experimental section we will explore the performance of four solution variants: The 'standard' method, which extracts the first t entries in $SA[sp,ep]$; a variant we call 'select', which enhances the baseline by using *rank* and *select* to first report all $SA[k]$ with $B[k]=1$; and the described RMQ approach on D', with the priority queue ordering according to the minimum value $D[k]$ ('RMQ') or the expected minimum in the intervals ('RMQ+est.min.').

Locating occurrences in text position order. By giving distinct semantics to the D array, we can use the same RMQ-based mechanism to prioritize the extraction of the occurrences in different ways. An immediate application, already proposed in the literature (but not implemented) [33], is to report the occurrences in text position order, that is, using $D[k] = SA[k]$. In the experimental section we show that our implementation of this mechanism is faster than obtaining all the $SA[sp,ep]$ values and sorting them, even when a significant fraction of the occurrences is to be reported.

6. Range-Restricted Location of Occurrences

We now extend the mechanism of the previous section to address, partially, the more complex problem of retrieving the occurrences of $SA[sp,ep]$ that are within a text window $T[l,r]$. Again, we focus on retrieving some of those "valid" occurrences, not all of them. We cannot guarantee a worst-case complexity (as it would not be possible in succinct space [21]), but expect that in practice we perform faster than the baseline of scanning the values left to right and reporting those that are within the range, $l \leq SA[k] \leq r$, until reporting the desired number of occurrences.

If, as in the end of Section 5, we obtain the occurrences in increasing text position order, we will eventually report the leftmost occurrence within $T[l,r]$, and since then we will report all valid occurrences. As soon as we report the first occurrence position larger than r, we can stop. Although introducing ordering in the process, this mechanism is unlikely to be very fast, because it must traverse all the positions to the left of l before reaching any valid occurrence.

We propose the following heuristic modification, in order to arrive faster to the valid occurrences. We again store tuples $\langle lp, rp, k, SA[k]\rangle$, where k gives the minimum position in $SA[lp,rp]$. But now we use a max-priority queue sorted according to $SA[k]$, that is, it will retrieve first the *largest* minima of

the enqueued ranges. After inserting the first tuple as before, we iteratively extract tuples $\langle lp, rp, k, SA[k]\rangle$. If $SA[k] > r$, then the extracted range can be discarded and we continue. If $SA[k] < l$, then we split the interval into two as before and reinsert both halves in the queue (position $SA[k]$ is not reported). Finally, if $l \leq SA[k] \leq r$, we run the algorithm at the end of Section 5 on the interval $SA[lp, rp]$, which will give all valid positions to report. This process on $SA[lp, rp]$ finishes when we extract the first value larger than r, at which point this segment is discarded and we continue the process with the main priority queue.

Note that, although this heuristic is weaker in letting us know when we can stop, it is likely to reach valid values to report sooner than using the algorithm of Section 5. In the experimental section we will show that, although our technique is slower than the baseline for general intervals (e.g., near the middle of the text), it is faster when the desired interval is close to the beginning (or the end, as desired). This biased range-restricted searching is useful, for example, in versioned systems, where the latest versions are those most frequently queried.

7. Experimental Results

All experiments were run on a server equipped with 144 GB of RAM and two Intel Xeon E5640 processors each with a 12 MB L3 cache. We used the Pizza&Chili corpus[3], which contains texts from various application domains.

Our implementations are based on structures of version 2.0.2 of sdsl. The CSAs of sdsl can be parameterized with the described traditional sampling method, which uses a bitmap B to mark the sampled suffixes. It has recently been shown [15] that this sampling strategy, when B is represented as `sd_vector` [34], gives better time/space tradeoffs than a strategy that does not use B but samples every $SA[i]$ with $i \equiv 0 \mod s$.

7.1. Combined sampling structures for locating and displaying

In our first experiment, we compare the traditional sampling using SA_s, SA_s^{-1} and B to the new solution that replaces SA_s^{-1} by just $\epsilon \cdot n/s$ samples plus a bitmap B' of length n/s to mark those samples in SA_s. We opted for $\epsilon = 1/8$, so that every SA^{-1} value can be retrieved in at most 8 steps, and B' is represented as an uncompressed bitmap (`bit_vector`). The underlaying CSA is a very compact FM-index (`csa_wt`) parameterized with a Huffman-shaped wavelet tree (`wt_huff`) and a compressed bitmap (`rrr_vector`). We choose this FM-index deliberately, since the ratio of space assigned to samples is especially high. With the new method we save so much space that we can also afford to represent B as `bit_vector`. Fig. 2 shows the time/space tradeoffs for accessing SA and SA^{-1} on one text (the results were similar on the others). The points representing the same s value in the new solution lie to the left left, since we

[3]Available under http://pizzachili.dcc.uchile.cl/.

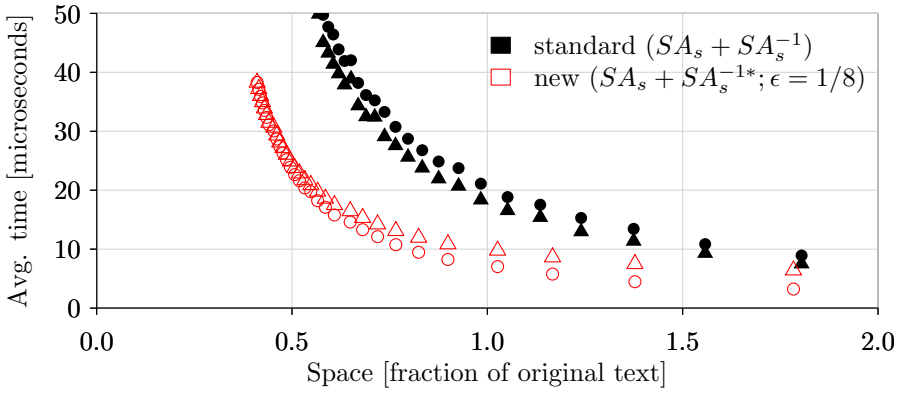

Figure 2: Time/space tradeoffs to extract a SA (○) respectively SA^{-1} value (△) from indexes over input `english.2108MB`. Sampling density s was varied between 1 and 32.

saved space. Replacing B with an uncompressed bitmap slightly reduces access time, in addition. We only report this basic experiment, but we note that these better tradeoffs directly transfer to applications that need simultaneous access to SA and SA^{-1}, like the child operation in compressed suffix trees.

7.2. Improved Ψ array representations

In the second experiment, we compare the performance impact of different Ψ encoding methods. We compare Elias-δ (Elias-γ performance is similar), `OptPFor` provided by the FastPFor library [26] and our own modified version of partitioned Elias-Fano encoding [35]. All encoding schemes encode blocks of size 128 and store starting positions of each block to allow pseudo-random access to Ψ. The performance of the Elias-δ scheme is optimized by using lookup tables and partial block decoding and to our knowledge represents a state-of-the-art baseline [22]. We additionally implemented the Uniform Block optimization in our `OptPFor` encoding scheme. As the alphabet size can influence decoding performance, we perform our experiment on `english.2108MB` from the Pizza&Chili corpus and on a word parsing `enwiki.4646MB` of the English Wikipedia ($\sigma = 3{,}903{,}703$).

Figure 3 shows the performance of accessing $\Psi[i]$ over both collections averaged over ten million random positions in $[0, n-1]$. No suffix array sampling is used in this experiment, as it does not affect the performance of $\Psi[i]$. As it can be seen, Partitioned Elias-Fano offers the best time performance, by a significant margin, but it is also slightly larger than the most space-economical alternative in each case. The difference in space is around 5%-10%, however, whereas the time difference is more significant. In the byte alphabet case, the Elias-Fano scheme can answer $\Psi[i]$ queries in around 200 nanoseconds, which is roughly the cost of two memory accesses and thus close to optimal.

Interestingly, `OptPFor` dominates the classical Elias-δ in space and time on the larger alphabet size of `enwiki.4646MB` (being the difference in time

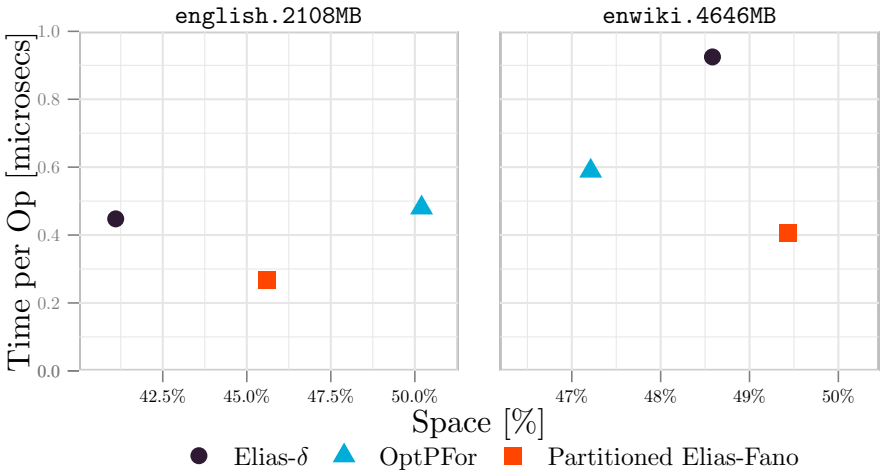

Figure 3: Mean time for one $\Psi[i]$ access in `english.2108MB` (left) and `enwiki.4646MB` (right) averaged over ten million operations for sample rate 16 and different Ψ encoding schemes.

more significant), whereas the opposite occurs on the smaller alphabet size of `english.2108MB` (being the difference in space more significant).

We conjecture that the better performance of `OptPFor` in `enwiki.4646MB` is a result of the higher number of run transitions within blocks of Ψ induced by the larger alphabet size. These transitions are encoded as large positive numbers, which can be efficiently handled by PForDelta encodings, whereas more work during decoding and/or more bits of space are necessary on the other encodings.

Next, we measure the impact of improved $\Psi[i]$ access performance on the speed of $SA[i]$ and $SA^{-1}[i]$ operations. Figure 4 shows the mean access time for both operations, again averaged over ten million executions. Our new sampling method using sample rates in $\{8, 16, 32, 64\}$ and the *`sd_vector`* is used in this experiment to achieve different time-space trade-offs. The run-time performance is similar to what is observed for $\Psi[i]$ access. The `OptPFor` scheme significantly outperforms the Elias-δ scheme for integer alphabets, but performs slightly worse on byte alphabets. Again, the Elias-Fano scheme outperforms both other schemes in all test instances, for the same sampling rate. Compared to Elias-δ encoding, it is almost twice as fast for large sampling rates, yet the difference diminishes as the sampling becomes denser. Still, the difference in run time performance is not as significant as for isolated $\Psi[i]$ access. This is likely caused by the additional access to the *`sd_vector`* each time Ψ is accessed, which can incur a non-negligible speed penalty.

The space usage also varies between test instances. The `OptPFor` encoding performs well for large alphabets, where it uses less space than the other two schemes. This is because many run transitions are encoded as exceptions, and the uniform block scheme further reduces the space usage of the encoding. For `english.2108MB`, however, `OptPFor` is outperformed in space by both other

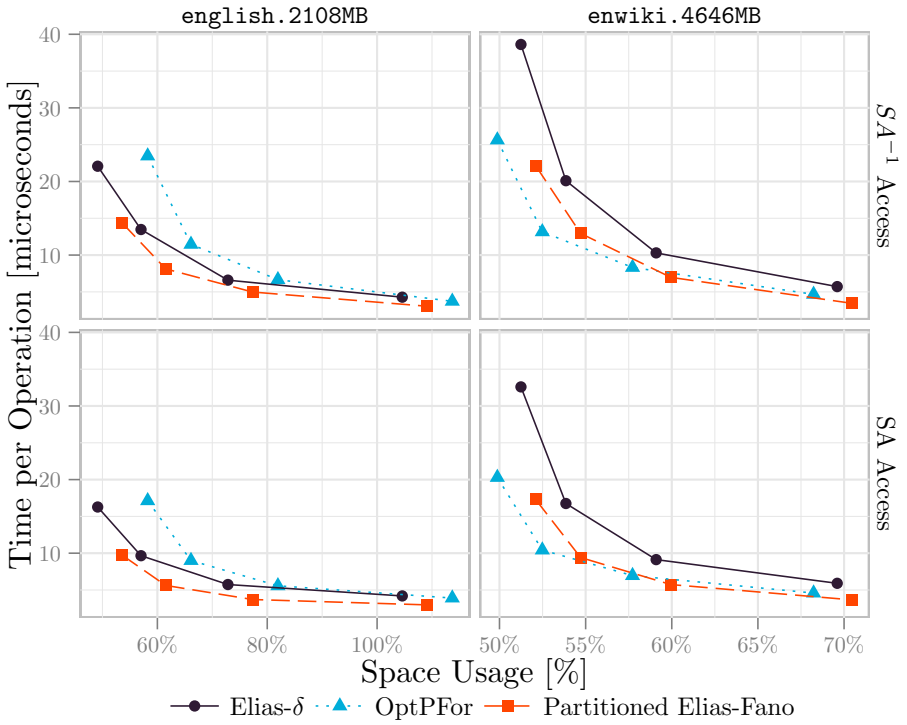

Figure 4: Mean time for one access to $SA^{-1}[i]$ (top) and to $SA[i]$ (bottom) in `english.2108MB` (left) and `enwiki.4646MB` (right) averaged over ten million operations for sample rates $8, 16, 32, 64$ and different Ψ encoding schemes.

schemes. In this input, the original Elias-δ encoding still obtains the best space performance.

If we consider both space and time, where differences in time can be traded for space by using sparser or denser samplings, we have that, essentially, `OptPFor` is the dominating alternative for large alphabets, whereas the Elias-Fano scheme dominates on byte alphabets.

7.3. Prioritized location of occurrences

In the third experiment, we measure the time to extract $t = 50$ arbitrary $SA[k]$ values from a range $[sp, ep]$. We use the same FM-index of Section 7.1. We create one index with $s = 6$ and another using an RMQ structure on D' (`rmq_succinct_sct`). Setting $s' = 10$ for the latter index results in a size of 2,261 MB, slightly smaller than the $s = 6$ index (2,303 MB).

Fig. 5 (top) shows the time and the average distance of the retrieved values to their nearest sample. Solution 'standard' spends the expected $(s'-1)/2 = 2.5$ LF steps per SA value, independently of the range size. Method 'select' first reports all sampled values in the range, hence the average distance linearly decreases and is close to zero at $s \times 50 = 300$. The RMQ based indexes spend

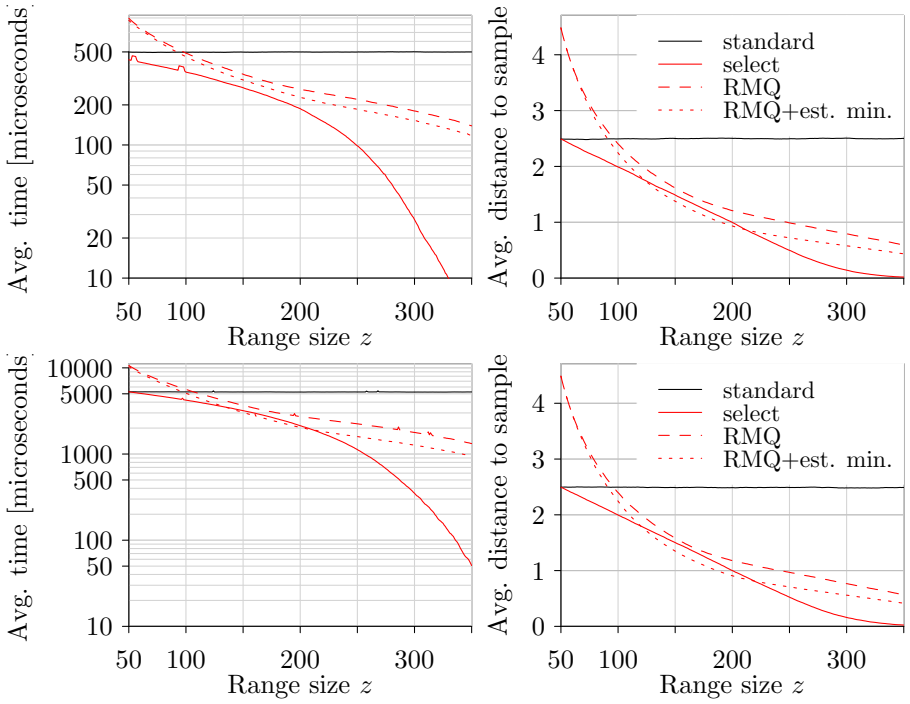

Figure 5: Left: Time to report 50 values in the range $SA[sp, sp+z-1]$. Right: Distance of a reported $SA[k]$ value to its nearest sample. Input: `english.2108MB` (top) and `enwiki.4646MB` (bottom).

the expected $(s-1)/2 = 4.5$ steps for $z = 50$. Using the RMQ information helps to decrease time and distance faster than linearly. The version using minimum estimation performs fewer LF steps for ranges in $[150, 220]$, but the cost of the RMQs in this case is too high compared to the saved LF steps.

In scenarios where LF is more expensive, 'RMQ+est.min.' can also outperform 'select' in runtime. The cost of one LF step depends logarithmically on the alphabet size σ, while the RMQ cost stays the same. Thus, using a text over a larger alphabet yields a stronger correlation between the distance and runtime, as shown in Fig. 5 (bottom), where we repeat the experiment using an FM-index (`csa_wt` parameterized with `wt_int`) on `enwiki.4646MB` . The RMQ supported index takes 3362 MB for $s' = 10$ and we get 3393 MB for $s = 6$.

Using almost the same index on `english.2108MB` (RMQ is built on SA this time, using $s' = 32$ and $s = 10$, obtaining sizes 1,554 MB and 1,590 MB), we now evaluate how long it takes to report the $t = 10$ smallest $SA[k]$ values in a range $SA[sp, ep]$. The standard version sequentially extracts all values in $SA[sp, ep]$, while keeping a max-priority queue of size k with the minima. The RMQ based method uses a min-priority queue that is populated with ranges and corresponding minimum values. Fig. 6 contains the results. For range size $z = 10$, the standard method is about 3 times faster, since we decode 10 values

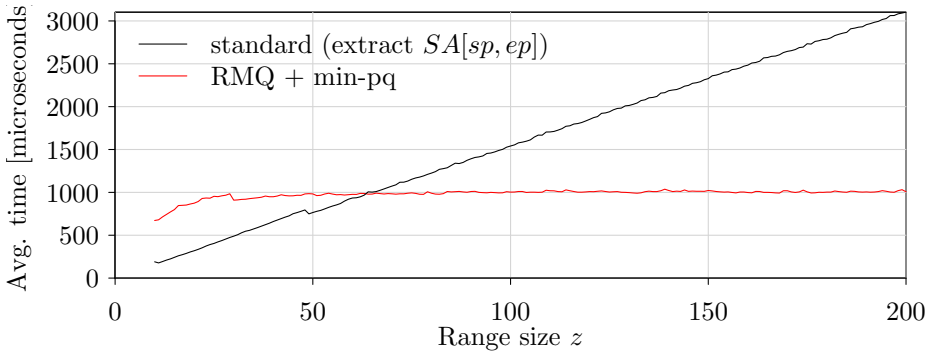

Figure 6: Average time to report the ten smallest $SA[k]$ values in $SA[sp, sp+z-1]$.

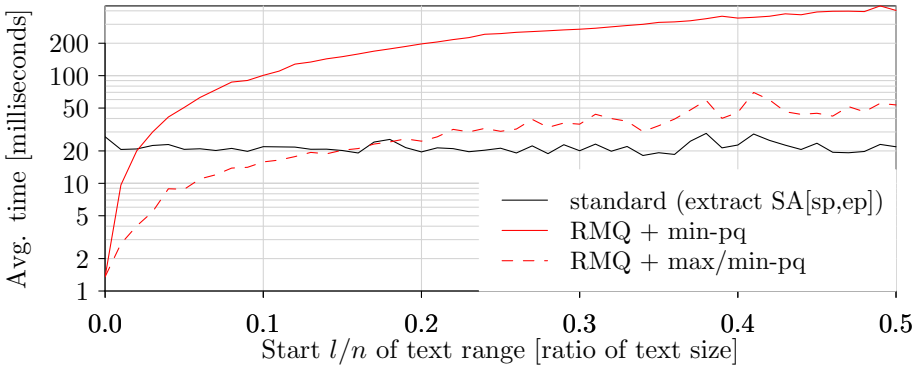

Figure 7: Average time to report ten $SA[k] \in [l, r]$ for k in $[sp, sp+10000-1]$.

in both methods and the sampling of the standard method is 3.2 times denser than that of the RMQ supported index. The RMQ index extracts $2t-1$ values in the worst case, when there are $t-1$ left in the priority queue. Therefore it is not surprising that the crossing point lies at about $60 \approx 3.2 \times (2t-1)$.

Lastly, we explore the performance of range-restricted locating on the same indexes. We take pattern ranges of size 10,000 and search for occurrences in text ranges $T[l, l+0.01n]$, which corresponds to the scenario drawn earlier in the paper. Fig. 7 shows that our heuristic using a max-priority queue to retrieve subranges that contain values $\geq l$, is superior to the standard approach in the first 15% of the text. The approach of the previous experiment extracts first all the occurrences located in $T[0, l-1)$, and thus becomes quickly slower than the standard approach.

References

[1] D. Belazzougui and G. Navarro. Alphabet-independent compressed text indexing. In *Proc. ESA*, pages 748–759, 2011.

[2] P. Bille and I. Gørtz. Substring range reporting. In *Proc. CPM*, LNCS 6661, 2011.

[3] M. Burrows and D. Wheeler. A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, 1994.

[4] P. Elias. Efficient storage and retrieval by content and address of static files. *J. ACM*, 21(2):246–260, 1974.

[5] P. Ferragina, R. González, G. Navarro, and R. Venturini. Compressed text indexes: From theory to practice. *ACM J. Exp. Alg.*, 13:art. 12, 2009.

[6] P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *Proc. FOCS*, pages 390–398, 2000.

[7] P. Ferragina and G. Manzini. Indexing compressed texts. *J. ACM*, 52(4):552–581, 2005.

[8] P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro. Compressed representations of sequences and full-text indexes. *ACM Trans. Alg.*, 3(2):article 20, 2007.

[9] P. Ferragina, I. Nitto, and R. Venturini. On optimally partitioning a text to improve its compression. *Algorithmica*, pages 51–74, 2011.

[10] P. Ferragina, J. Sirén, and R. Venturini. Distribution-aware compressed full-text indexes. *Algorithmica*, 67(4):529–546, 2013.

[11] J. Fischer and V. Heun. Space-efficient preprocessing schemes for range minimum queries on static arrays. *SIAM J. Comp.*, 40(2):465–492, 2011.

[12] J. Fischer, V. Mäkinen, and G. Navarro. Faster entropy-bounded compressed suffix trees. *Theor. Comp. Sci.*, 410(51):5354–5364, 2009.

[13] S. Gog, T. Beller, A. Moffat, and M. Petri. From theory to practice: Plug and play with succinct data structures. In *Proc. SEA*, LNCS 8504, pages 326–337, 2014.

[14] S. Gog and G. Navarro. Improved and extended locating functionality on compressed suffix arrays. In *Proc. SEA*, LNCS 8504, pages 436–447, 2014.

[15] S. Gog and M. Petri. Optimized succinct data structures for massive data. *Soft. Prac. & Exp.*, 44(11):1287–1314, 2014.

[16] G. Gonnet, R. Baeza-Yates, and T. Snider. *Information Retrieval: Data Structures and Algorithms*, chapter 3: New indices for text: Pat trees and Pat arrays, pages 66–82. Prentice-Hall, 1992.

[17] R. González, G. Navarro, and H. Ferrada. Locally compressed suffix arrays. *ACM Journal of Experimental Algorithmics*, 19(1):article 1, 2014.

[18] R. Grossi, A. Gupta, and J. S. Vitter. High-order entropy-compressed text indexes. In *Proc. SODA*, pages 636–645, 2003.

[19] R. Grossi and J. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. In *Proc. STOC*, pages 397–406, 2000.

[20] R. Grossi and J. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM J. Comp.*, 35(2):378–407, 2006.

[21] W.-K. Hon, R. Shah, S. . Thankachan, and J. Vitter. On position restricted substring searching in succinct space. *J. Discr. Alg.*, 17:109–114, 2012.

[22] H. Huo, L. Chen, J. S. Vitter, and Y. Nekrich. A practical implementation of compressed suffix arrays with applications to self-indexing. In *Proc. DCC*, pages 292–301, 2014.

[23] J. Kärkkäinen, P. Sanders, and S. Burkhardt. Linear work suffix array construction. *J. ACM*, 53(6):918–936, 2006.

[24] D. Kim, J. Sim, H. Park, and K. Park. Constructing suffix arrays in linear time. *J. Discr. Alg.*, 3(2–4):126–142, 2005.

[25] P. Ko and S. Aluru. Space efficient linear time construction of suffix arrays. *J. Discr. Alg.*, 3(2–4):143–156, 2005.

[26] D. Lemire and L. Boytsov. Decoding billions of integers per second through vectorization. *Software: Practice and Experience*, 45(1):1–29, 2015.

[27] V. Mäkinen and G. Navarro. Rank and select revisited and extended. *Theor. Comp. Sci.*, 387(3):332–347, 2007.

[28] U. Manber and G. Myers. Suffix arrays: a new method for on-line string searches. *SIAM J. Comp.*, 22(5):935–948, 1993.

[29] G. Manzini. An analysis of the Burrows-Wheeler transform. *J. ACM*, 48(3):407–430, 2001.

[30] J. Munro, R. Raman, V. Raman, and S. Rao. Succinct representations of permutations. In *Proc. ICALP*, LNCS n. 2719, pages 345–356, 2003.

[31] S. Muthukrishnan. Efficient algorithms for document retrieval problems. In *Proc. SODA*, pages 657–666, 2002.

[32] G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Comp. Surv.*, 39(1):art. 2, 2007.

[33] Y. Nekrich and G. Navarro. Sorted range reporting. In *Proc. SWAT*, LNCS 7357, pages 271–282, 2012.

[34] D. Okanohara and K. Sadakane. Practical entropy-compressed rank/select dictionary. In *Proc. ALENEX*. SIAM, 2007.

[35] G. Ottaviano and R. Venturini. Partitioned Elias-Fano indexes. In *SIGIR*, pages 273–282, 2014.

[36] R. Raman, V. Raman, and S.S. Rao. Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. *ACM Trans. Alg.*, 3(4):art. 43, 2007.

[37] S. Rao. Time-space trade-offs for compressed suffix arrays. *Inf. Proc. Lett.*, 82(6):307–311, 2002.

[38] K. Sadakane. Compressed text databases with efficient query algorithms based on the compressed suffix array. In *Proc. ISAAC*, LNCS v. 1969, pages 410–421, 2000.

[39] K. Sadakane. New text indexing functionalities of the compressed suffix arrays. *J. Alg.*, 48(2):294–313, 2003.

[40] K. Sadakane. Compressed suffix trees with full functionality. *Theor. Comp. Sys.*, 41(4):589–607, 2007.

[41] K. Sadakane. Succinct data structures for flexible text retrieval systems. *J. Disc. Alg.*, 5(1):12–22, 2007.

[42] S. Vigna. Quasi-succinct indices. In *Proc. 6th International Conference on Web Search and Data Mining (WSDM)*, pages 83–92, 2013.

[43] P. Weiner. Linear pattern matching algorithm. In *Proc. 14th IEEE Symposium on Switching and Automata Theory*, pages 1–11, 1973.

[44] H. Yan, S. Ding, and T. Suel. Inverted index compression and query processing with optimized document ordering. In *Proc. International World Wide Web Conference (WWW)*, pages 401–410. ACM, 2009.

[45] J. Zhang, X. Long, and T. Suel. Performance of compressed inverted list caching in search engines. In *Proc. International World Wide Web Conference (WWW)*, pages 387–396, 2008.